(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
23 August 2001 (23.08.2001)

PCT

(10) International Publication Number
WO 01/61486 A2

(51) International Patent Classification[7]: G06F 9/445

(21) International Application Number: PCT/US01/05409

(22) International Filing Date: 20 February 2001 (20.02.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/505,919 17 February 2000 (17.02.2000) US

(71) Applicant *(for all designated States except US)*: **INTERACTIVE VIDEO TECHNOLOGIES, INC.** [US/US]; 10940 Wilshire Boulevard, Suite 900, Los Angeles, CA 90024 (US).

(71) Applicant and
(72) Inventor: **WASON, Andrew** [AU/US]; 6 Victorian Woods Drive, Atlantic Highland, NJ 07716 (US).

(74) Agents: **BIERNACKI, John, V.** et al.; Jones, Day, Reavis & Pogue, North Point, 901 Lakeside Avenue, Cleveland, OH 44114 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

WO 01/61486 A2

(54) Title: INCREMENTAL BOOTSTRAP CLASS LOADER

(57) Abstract: This is a method for incremental downloading of a software application with several components. First, a small stub module is downloaded. The stub module then downloads some subset of the application's components sufficient to launch the application. While the application runs, the need for additional components is detected. If the required components are unavailable locally, there are downloaded from a file server, and the locally maintained listing of available components is updated accordingly. The method described can also be used to update a software release in a similar, incremental manner.

INCREMENTAL BOOTSTRAP CLASS LOADER

TECHNICAL FIELD

This invention relates to methods for remote software downloading and updating.

BACKGROUND OF THE INVENTION

Software applications usually are built in discrete modules. Many reasons for the modular construction exist; because the reasons are well known for the most part, we mention only one, and a very general one at that: structured programming.

Software is often downloaded through a network into a machine that performs actual execution. The network can be a local area network ("LAN") or a wide area network ("WAN"). Although download speeds have improved considerably over the past several years, an average application's size has grown even faster than the download speeds. As an example, downloading over the Internet — the most widely used network — through a conventional analog telephone line is performed at speeds of only several megabytes per hour.

At the same time, the number of software downloads is growing because of several factors, including the following:

1. Downloading is quickly becoming the preferred means for software distribution;

2. Proliferation of personal computer use;

3. Growth of telecommuting;

4. Expansion in the number of software publishers;

5. Shorter period between software releases.

License administration considerations and local storage limitations also favor downloading, although these aspects are less likely to affect a typical non LAN-based individual user.

In sum, software downloads are becoming more common and more time consuming. It is therefore desirable to reduce the download size. It is particularly desirable to reduce the size of the initial download, *i.e.*, the size of the code that must be downloaded before a user can launch an application for the first time.

One known attempt to reduce download size is by defining "profiles" - a specification of the minimum set of Application Programming Interfaces ("APIs") useful for a particular kind of application. Because of their generality, these profiles, will often include functionality a given application does not need, and will not include functionality that another application may require. One size indeed does not fit all.

OBJECT OF THE INVENTION

The object of this invention is to provide a faster method for incremental software downloading by decreasing initial download size.

SUMMARY OF THE INVENTION

To achieve this and other objects, this invention initially loads an incremental bootstrap class loader - a small module that controls further loading of a software application. The incremental loader proceeds to load all software modules that are required to launch the application. As a matter of practical experience, the initially required components can be, and usually are, much smaller than the total size of the application. Thus, in accordance with this invention, the user can begin using the application much sooner than if the user loaded the entire application before launching it.

After the initial transfer, the incremental loader automatically senses the need for additional modules, connects to the application's file server, obtains the additional modules, and stores them locally.

Other features and advantages of the present invention will become apparent from the following description taken in conjunction with the code listed in the accompanying appendix.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT OF INVENTION

In developing the preferred embodiment, our goal was to reduce the initial download size of a JavaTM Virtual Machine ("JVM") without losing JVM functionality. JVM is a byte-code interpreter for executing code written in Java™, an

architecture-neutral, object-oriented, multithreaded language intended for use in distributed environments.

Simply removing pieces of the JVM, *e.g.*, the GUI libraries, does reduce the download size; it also causes loss of JVM functionality. But every JVM is required to contain a specified set of functional capabilities to be "Java" compatible. The set of functional capabilities is specified in the JavaTM Compatibility Kit ("JCK"), a set of test cases.

The JVM thus includes the core virtual machine, a large set of core Java™ class files, and a set of native libraries — dynamic link libraries ("DLLs") on Win32 platforms — that embody the functionality of the core Java™ classes, with the native libraries containing the "native method"- specific implementations. The "core" JVM components are required for Java™ compatibility. Additional functions and classes with their associated native code may be used by application developers to extend the functionality of the platform. We shall refer to these functions and classes as "Extensions." Extensions may be installed with a particular JVM. Extensions are not required for Java™ compatibility.

In the course of running a specific Java™ application, the JVM loads Java™ classes and native libraries from the local disk into program memory. Core classes and their associated native libraries are loaded using a bootstrap class loader. The bootstrap class loader loads only those classes and native libraries that are actually required for the functioning of the specific application. As the application runs,

additional classes and libraries may be dynamically loaded as they are needed. For example, if the user pulls down a menu in the graphical user interface, the class loader will dynamically load and cache the *Menu* and *MenuItem* classes.

To reduce the initial download size, we download a small stub JVM with our incremental class loader. The incremental class loader recognizes requests for core classes and native libraries that have not been cached locally, downloads them from a specified location on the network, and permanently stores them on the local disk, so that future requests for these components can be resolved locally.

This mechanism downloads the JVM piece-by-piece. Only what is used is downloaded. Typically, this results in a substantial download size saving for the first application, and subsequent application downloads are even smaller because the first application uses many common components.

Our incremental class loader uses standard compression techniques. Additionally, it eliminates superfluous Constant Pool entries in Java™ classes. Each Java™ class file includes a Constant Pool, *i.e.*, a list of all constants referenced by the class. Many Constant Pool entries are replicated across the thousands of core Java™ classes. All of the core Java™ class Constant Pools can be extracted into a single Global Constant Pool with no duplicate entries. The individual class Constant Pool entries can be replaced with indices into the Global Constant Pool. This results in a highly compressed Java™ class file even before invocation of standard compression techniques.

The incremental class loader keeps track of the downloaded files by modifying "jar" files that contain Java™ classes. A minimal version of these jar files is included in the initially downloaded stub JVM. Each file has an empty JarEntry structure for each class file belonging to the jar file. (Description of the JarEntry class can be found in http://www.javasoft.com/products/jdk/1.2/docs/api/javas/util/jar/JarEntry.html.) The collection of JarEntries acts as a manifest specifying the set of core Java™ class files that can be remotely downloaded. When the incremental class loader receives a request to load a specified class, it loads that class from the jar file. If the class length is zero, then the class loader must download the class from the remote class file server, uncompress it, and rebuild its Constant Pool by replacing the indices into the Global Constant Pool with the actual entry values from the Global Constant Pool.

Initially, the Global Constant Pool is also empty. When a class is downloaded, each index it contains into the Global Constant Pool will cause the index's associated entry to be downloaded and placed in the Global Constant Pool.

All downloaded components must be verified to prevent installation of malicious code or data on the user's system. This can be done by computing a hash of the data, *e.g.*, a checksum or a CRC, and encrypting the hash using a private key. This encrypted hash is sent over the network along with the data. The corresponding public key is included in the stub JVM download. Before using the downloaded data, the incremental class loader computes its hash, decrypts the downloaded hash using its public key, and compares the decrypted hash to the computed hash. If the two hashes

match, the data is safe to use. Otherwise, it has been corrupted and an appropriate exception is processed.

To summarize, when the incremental class loader receives a request for a class, it performs the following steps:

1. Load the JarEntry for the class from the local core jar file;
2. If the JarEntry has zero length,
   (i) Download the class file from the class file server,
   (ii) Uncompress the class file using standard compression techniques,
   (iii) Verify the class file as described above,
   (iv) Examine the class file's Constant Pool and, for each index that is not populated in the Global Constant Pool, download and verify the entry for that index, populating the Global Constant Pool,
   (v) Reconstitute the class's Constant Pool with entries from the Global Constant Pool,
   (vi) Store the class file in the corresponding JarEntry of a local jar file; and
3. Load the class entry into memory.

The procedure for native libraries is slightly different. A table of native library entries is maintained. When missing libraries are downloaded, they are marked in the

table as locally available. Native libraries are compressed using standard techniques and they are verified.

Because multiple processes may be using the JVM simultaneously, access to the local class file store must be properly synchronized. One way to synchronize access is to have a single class store manager process for managing the class file store. To download classes, all other processes using the JVM communicate with the class store manager via local sockets. The single class store manager process internally synchronizes its access to the class file store.

A variation on the incremental loading method described can be used to upgrade a JVM from one release to another. This requires "tagging" each jar file entry with a version ID. When a class is requested and the class is available locally, its ID is checked. A class with a current ID is loaded into memory. If a class's ID is not current, the class file is conditionally downloaded from the class file server. If the version on the class file server has changed, the new file is downloaded, stored, and tagged with the new ID. Otherwise, the file is locally re-tagged with the new version and not downloaded.

The methods described in this specification obviously can be applied to Installed Extensions and applications other than JVM. Moreover, those skilled in the art will be able to devise various modifications that although not explicitly described or shown herein, embody the principles of the invention and are thus within its spirit and scope.

I CLAIM:

1. A method for incrementally downloading a software application having a plurality of components, said method comprising the steps of:

downloading a subset of said plurality of components, the components of the subset being sufficient to launch said application;

launching said application;

sensing the application's requirement for components;

downloading a first additional component that has not been previously downloaded after the application requires said first additional component.

2. A method for downloading according to claim 1, further comprising the step of storing the downloaded components locally.

3. A method for downloading according to claim 1, further comprising the step of verifying integrity of the downloaded components.

4. A method for downloading according to claim 1, further comprising the steps of compressing one of the components before downloading said one of the components and uncompressing the downloaded one of the components.

5. A method for downloading according to claim 1, further comprising the steps of:

compressing the components before downloading;

uncompressing the downloaded components;

verifying integrity of the downloaded components; and

storing the downloaded components locally.

6. A method for downloading according to claim 5, wherein said step of downloading a subset of said plurality of components is a step for downloading a subset of said plurality of components, said step of downloading a first additional component is a step for downloading a first additional component, said step of sensing the application's requirement for components is a step for sensing the application's requirement for components, said step of compressing the components is a step for compressing the components, said step of uncompressing the downloaded components is a step for uncompressing the downloaded components, and said step of verifying integrity of the downloaded components is a step for verifying integrity of the downloaded components.

7. A method for downloading a JVM having a plurality of components, said plurality of components including a group of Java class files,

each Java class file of said group of Java class files being associated with a Java class, said each Java class file having a Constant Pool, said Constant Pool of said each Java class file including a list of constants referenced by the Java class associated with said each Java class file, said method comprising the steps of:

downloading a subset of said plurality of components, the components of the subset being sufficient to launch said JVM;

launching the JVM;

sensing the launched JVM's requirement for components; and

downloading additional components that have not been previously downloaded after the JVM requires the additional components;

wherein the subset and the additional components include a plurality of Java class files from the group of Java class files.

8. A method for downloading according to claim 7, further comprising the step of compressing the Java class files of said plurality of Java class files before downloading, and uncompressing the downloaded Java class files.

9. A method for downloading according to claim 8, wherein

said step of compressing Java class files includes compressing Java class files belonging to a first set;

said method further comprising the steps of:

extracting constants of Constant Pools of the Java class files belonging to the first set into a Global Constant Pool, said step of extracting performed before compressing the Java class files belonging to the first set;

replacing the constants of the Constant Pools of the Java class files belonging to the first set with corresponding indices into the Global Constant Pool;

downloading constants of the Global Constant Pool;

building a downloaded version of the Global Constant Pool; and

replacing the indices into the Global Constant Pool in the downloaded Java class files belonging to the first set with corresponding constants of the Global Constant Pool.

10. A method for downloading according to claim 9, further comprising the step of verifying integrity of the downloaded components.

11. A method for downloading according to claim 9, further comprising a step for verifying integrity of the downloaded components.

12. A method for downloading according to claim 9, wherein said step of downloading constants of the Global Constant Pool is a step for downloading constants of the Global Constant Pool.

13. A method for downloading according to claim 9, wherein a first Java class file having a first Constant Pool is downloaded, said first Constant Pool having a first constant that is not present in the downloaded version of the Global Constant Pool, further comprising the steps of:

downloading the first constant; and

populating the downloaded version of the Global Constant Pool with the first constant.

14. A method for downloading according to claim 9, further comprising the step of verifying the downloaded constants before said building step.

15. A method for running the JVM downloaded in accordance with claim 9, further comprising the steps of:

storing each downloaded component of the JVM locally; and

synchronizing access to said each downloaded component of the JVM.

16. A method for upgrading a software application having a plurality of components to a first version, said components being stored locally, said method comprising the steps of:

storing locally a plurality of tags, one tag per component, each tag indicating a current version of said each tag's associated component;

sensing the application's requirement for loading the components;

checking a first tag associated with a first component after the application requires said first component;

conditionally downloading from a file server the first version of the first component if the first tag indicates that the locally stored first component is older than the first version of the first component;

downloading the first version of the first component if the first version of the first component differs from the locally stored first component;

storing locally the downloaded first version of the first component; and

updating the first tag to indicate the first version.